The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, Month 2011, 1–29

RESEARCH ARTICLE

GPU parallelization strategies for metaheuristics: a survey

Mokhtar Essaid, Lhassane Idoumghar, Julien Lepagnot and Mathieu Brévilliers

Université de Haute-Alsace, LMIA (EA 3993), F-68100 Mulhouse, France

(*v3.6 released March 2011*)

Metaheuristics have been showing interesting results in solving hard optimization problems. However, they become limited in terms of effectiveness and runtime for high dimensional problems. Thanks to the independency of metaheuristics components, parallel computing appears as an attractive choice to reduce the execution time and to improve solution quality. By exploiting the increasing performance and programability of graphics processing units (GPUs) to this aim, GPU-based parallel metaheuristics have been implemented using different designs. Recent results in this area show that GPUs tend to be effective co-processors for leveraging complex optimization problems. In this survey, mechanisms involved in GPU programming for implementing parallel metaheuristics are presented and discussed through a study of relevant research papers.

Keywords: metaheuristics; optimization problems; GPU; survey

1. Introduction

The objective of optimization problems is to find an optimal solution among all the feasible solutions in a given search space. They are classified as continuous or combinatorial, constrained or unconstrained, single or multi-objective, static or dynamic [14]. Their importance rise in several areas such as industry, management, engineering, networks, etc [73]. Often, for an adressed optimization problem, a solution of good quality has to be quickly obtained. To cover this issue, metaheuristics have been proposed to give near optimal solutions with the respect of a reasonable time.

However, most of metaheuristics suffer from the lack of scalability: the performance decreases in both terms of time complexity and effectiveness when facing high dimensional problems. To overcome this limit, GPU-based parallel metaheuristics have been proposed. This latter has attracted a growing interest from the scientific community in order to reduce the execution time and to improve the quality of the solutions found. According to [73], the parallel design of metaheuristics is classified into three classes:

(1) Algorithmic level: this level allows the run of many algorithms in parallel. The algorithms can run independently with different starting solutions and/or different parameters and choose the best results of the run. In this case, the result will be the same as if we run all these algorithms sequentially, i.e. we reduce the execution time. The algorithms can also cooperate with each other which means that the behavior of the metaheuristics can change and improve the quality of resulting solutions.

*Corresponding author. Email: mokhtar.essaid@uha.fr

ISSN: 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/17445760.YYYY.CATSid
http://www.informaworld.com

(2) Iteration level: this level allows a parallelization in each iteration. This is a parallelization of the evaluations and/or the generation of the neighborhood. Different parts of the neighborhood are executed in parallel. The behavior of the metaheuristic is not altered. The main objective is to speed up the algorithm by reducing the search time.
(3) Solution level: this level allows a parallelization of a single solution. For example, to evaluate the objective function or constraints for a generated solution. The behavior of the metaheuristic is not altered. The objective is mainly the speed up of the search.

With different designs, many parallel algorithms have been successfully implemented using GPUs. Nowadays, the GPUs have become a powerful tool in high performance computing. They rely on the single-instruction-multiple-data (SIMD) architecture which supports a parallel execution of hundreds of threads at the same time. This survey aims to complement previous reviews which covered the use of GPUs to implement nature-inspired metaheuristics [48], and described different parallelism strategies and communication patterns of metaheuristics on GPUs [9]. Moreover, it uses a different organization based on the metaheuristics classification to allow covering more works.
The rest of the paper is organized as follows. Section 2 presents the basic concepts of GPU computing. Section 3 describes implementations of parallel metaheuristics using GPUs to solve combinatorial and continuous problems, which are discussed in Section 4. Finally, concluding guidelines to implement an efficient parallel algorithm are presented in Section 5.

2. GPU computing

GPUs have achieved great performances in the last years. This modern hardware has been mainly designed to support video games and 3D graphic applications [17], but subsequently, it has been employed for general computational purposes. It is now used in various fields such as data compression [59], image processing [87], data mining [33], etc. The availability of application programming interfaces (API) has eased implementing parallel applications, but there is still a need to follow some crucial principles in order to get efficient performance from GPUs.
The understanding of the underlying parallel programming model of GPU is necessary to design a parallel application (in our case metaheuristics). The *kernel* is the fundamental unit for a parallel application, by which the execution is conducted. It is a piece of code called from the CPU (also called the host) and duplicated on the GPU (also called the device). The *kernel* is run within a *grid* (set of *blocks*) where each *block* is a set of *threads* [25].
The memory management is a crucial aspect and represents a challenge when dealing with GPU based applications [47]. The memory architecture within Nvidia architecture consists of six types of memory. The first is the global memory which is the largest memory but its frequent exploitation affects negatively the performance. As a result, it is recommended to avoid its use as much as possible. The second is the constant memory. It is used when all the threads have the same space address. The third type is the shared memory. It is an on-chip memory of 64KB and used to share data between threads within a single block [46] . Thanks to its low latency, the shared memory can positively affect the efficiency when it is used properly. The last memory is the local memory which is the fastest but also the smallest. Each thread has its own local memory that can not be used by other threads. For further details about the Nvidia GPU architecture, we refer the reader to [58] [46] [47].

3. GPU architecture for parallel metaheuristics

3.1. Solving combinatorial problems using GPU-accelerated metaheuristics

This section reviews GPU-accelerated metaheuristics for solving combinatorial problems. We distinguish between metaheuristics according to their classification (single solution/population-based) [13] regarding the different designs that have been proposed for their implementations.

3.1.1. Single solution based metaheuristics

In this subsection, we outline different implementations for single solution based metaheuristics adressed to solve combinatorial problems. Single solution based metaheuristics start with one initial solution and generate a large neighborhood to pick the next solution according to a selection mechanism. We conclude the subsection by presenting Table 1 that summarizes the results and characteristics of these implementations.

Tabu Search (TS) is a single solution based metaheuristic that takes an initial solution to a problem and checks its neighbors looking for an improved solution [37]. Moreover, tabu search uses a short term memory to store the visited solutions, or some of their attributes, in order to avoid coming back to previously visited solutions [39].
From the literature, we mention [71] where a parallel multi-objective version of TS is proposed to tackle multi-criteria vehicle routing problem (MVRP), using an external archive of non dominated solutions. This algorithm performs a number of parallel local searches with different starting points from solutions of current Pareto front (algorithmic level). Solutions are evaluated and compared with their predecessors: if the new solution is better than the old one, then it takes its place in next iteration, else the algorithm ignores the solution and the search continues with the current solution. A new solution is added to the Pareto front if it dominates the solutions of the front. Their results show that the parallel version of the algorithm is faster than the sequential version by 14 times and it explores over 16 times more neighborhoods.

In 2013, a parallel multi start tabu search has been proposed to solve the quadratic assignment problem (QAP) [27]. The design presents two level of parallelism (algorithmic level/iteration level). It generates initial solutions from which parallel tabu search instances are run (algorithmic level). Then, a parallel neighborhood generation is performed for each instance (iteration level) because each permutation in the neighborhood can be evaluated independently. In [27], the tabu list is represented as an array, in which a value equal to zero indicates that the respective move is not tabu, while a positive value indicates how many iterations the move will stay in the list. The array of accepted moves is divided into blocks, and the best move is found in each block by the tournament method. This method is applied again to find the final minimum value (best move selection). The authors have noticed the impact of an appropriate memory management (coalesced memory or non coalesced memory), and the nature of the instance (symmetric or not) on neighborhood generation speedup.

In the same context, QAP has been addressed in [8] by proposing another version of iterative parallel tabu search (ITTSD), where different instances of TS are run in parallel (algorithmic level). Afterwards, a diversification strategy proposed in [38] is applied. This strategy is performed after each global iteration on the best global solution, in order to strengthen the exploration capability of the algorithm. In addition, if the TS instances could not improve the solutions quality after a

Table 1.: Single solution based metaheuristics

Ref	Problem	Algorithm	Parallelism level	Acceleration	Benchmark	Quality improvement	System CPU/GPU
[27]	QAP	TS	Algorithmic level+Iteration level	420x compared to CPU implementation and 70x faster than a parallel CPU implementation on a high-end six-core CPU	Randomly generated following the method proposed by [72] and QAPLIB [5]	Quality improved in long runs	CPU:six-core Intel Core i7-980x (3.33 GHz) / GPU: NVidia GTX 480 with 480 cores at 1401 MHz
[8]	QAP	TS	Algorithmic level + Iteration level + Solution level	GPU implementation saves up to 3 hours compared to CPU sequential implementation	QAPLIB [5] + instances taken from [31]	Quality improved with deviation between -3.61% and -31.98% compared to the best known results	CPU:Intel Core processor i5-3330 (3.00GHz) / GPU: NVIDIA GeForce GTX680 with 1536 cores at 1058 MHz
[71]	MVRP	TS	Algorithmic level	14x compared to CPU sequential implementation	Instances of size from 10 to 500	Quality improved (90 non dominated solutions found) compared to classic TS (3 non dominated solutions found)	CPU: sequential implementation on Tesla S2050 (1.55 GHz)/GPU: GeForce GTX 480 with 480 cores at 1006 MHz
[18]	RCPSP	TS	Algorithmic level + Iteration level	5.4x compared to parallel CPU and 22x compared to sequential CPU implementation	J30, J60, J90 and J120 from [4]	Quality improved compared to other TS implementations	CPU: AMD Phenom II X4 945 server (3 GHz) /GPU: NVidia GeForce GTX 650 Ti with 768 CUDA cores at 928 MHz
[30]	MKP	GRASP	Algorithmic level + Iteration level	No acceleration	ORLIB [12]	Quality improved compared to ACO and GA and slightly better compared to CPU sequential implementation (CPU sequential implementation outperformed for some instances)	Not mentioned
[82]	TSP	SA	Algorithmic level + Iteration level	14.84x compared to CPU implementation	9 instances taken from TSPLIB [7]	Improving the best known solution by 27% compared to CPU implementation	CPU: AMD A8-3870K (3.0 GHz)/GPU: NVIDIA GeForce GTX680 with 1536 CUDA cores at 1058 MHz

certain number of global iterations, the algorithm applies a perturbation factor on them to lead the search toward new regions. Because of the computational cost of an evaluation, the authors propose a parallel evaluation of neighborhood using delta matrix (iteration level). Each value in this matrix is the cost of one of the possible permutations (one permutation is considered to generate one neighbor). Two kernels are used: the first initializes the costs of neighborhoods within delta matrix, and the second applies the best move on the solution. Then, it updates the delta matrix. After the experimentation, the authors found out that their algorithm attains promising results in terms of quality, achieving 19 new better results than the best solutions found in the literature, and saving from 2.34 and 3.87 hours of computation compared to the CPU sequential implementation with an approximate speedup of 1%.

In 2015, a parallel TS has been proposed to solve the resource constrained project scheduling problem (RCPSP) [18]. In their design, a kernel is run in several blocks, and each block represents a TS instance (algorithmic level). In every TS instance, an initial solution is generated by finding the longest paths from activity 0 to all other activities. Then, it assembles activities with the same distance from activity 0 into levels (distance is defined by how many activities are performed before the considered activity starts). Afterwards, a feasible schedule (solution) can be created from these levels. At the parallel neighborhood generation phase (iteration level), neighbors are generated by a set of moves where a move is considered as a swap in the schedule. Infeasible moves that violate the precedence relations between activities are ignored to reduce the time-consuming evaluation

of the neighbors. The blocks (TS instances) communicate with each other using the global memory, and the co-operation is performed by exchanging solutions using a working set F which contains the best known solution so far. Solutions exchange occurs if the last read solution in F has not been improved for more than a certain number of iterations, or if the block has found an improvement of it. The algorithm achieves up to 5.4/22 times faster compared to the parallel/sequential CPU respectively, and 10.5/42.7 times faster in J90 benchmark instances.

A sequential and parallel greedy randomized adaptive search procedure (GRASP) algorithm has been presented in [30] to solve the 0-1 multidimensional knapsack problem (MKP) using CUDA. The CUDA based algorithm allows to expand the neighborhood search. It launches different threads to construct initial solutions distributed on the search space (algorithmic level). Afterwards, it executes parallel local search in order to improve them (iteration level). The CPU controls the iterations, and it gets the best solution at the last stage. The results have shown the effectiveness of the algorithm. It obtains better results compared to genetic algorithms (GA) and ant colony optimization (ACO). The parallel version shows better solutions thanks to the large neighborhood exploration but without improvement in execution time.

In the same year, simulating annealing (SA) has been mapped in [82] on GPU. The authors propose a parallel SA algorithm to solve TSP. In their design, the CPU generates solutions for each block thread where blocks represent SA instances (Algorithmic level). Each thread in the same block generates one neighbor in parallel (iteration level). Instead of taking the best neighbor solution, the algorithm picks a random neighbor solution chosen from the best solutions found (best solutions are retained by performing a reduction operation). In order to generate the neighborhood in parallel, each thread randomly chooses two cities, and performs a swap operation between them on the path specified by the current solution, then it calculates the new cost. The authors have noticed that it is time consuming to calculate the cost of the new path from the head to the tail. They have observed that the cost of the old and the new paths differ only for the line segments connecting to a or b. Instead of swapping the two cities a and b then computing the cost of new path, they compute only the difference between the new cost and the current cost based on the distances of a and b with the cities (a-1, a+1, b-1, b+1) that are in the path. The set of better neighbor solutions is generated by a parallel reduction operation. The algorithm picks one solution among them randomly to the next iteration. Finally, the best solution is retained by a reduction operation on the blocks. The algorithm improves the solution quality by 27% and reaches an acceleration factor of 14.84 compared to the sequential version.

3.1.2. Population-based and hybrid metaheuristics

Unlike single solution based metaheuristics, population-based metaheuristics generate an initial group/groups of individuals that evolve using certain operators defined by the algorithm. In this subsection, we outline some works that exploit GPUs power in implementing this kind of metaheuristics for combinatorial problems. Finally, the characteristics and the results obtained by these metaheuristics are summarized in Table 2.

Ant colony optimization (ACO):

Thanks to the independent nature of ACO, a major focus has conducted the scientific community to implement parallel versions of the algorithm in order to tackle several optimization problems. We mention from the literature [21] which

has stated that the traditional way of parallelizing tour construction in ACO is to run threads (threads represent ants) in parallel looking for the best tour that they can find. According to [21], this approach has many limitations. For instance, it requires a relatively low number of threads. Furthermore, it presents an unpredictable memory access pattern, and a great probability of divergence within the warp (threads take different paths). In [21], an alternative design for solving Traveling Salesman Problem (TSP) is proposed. Besides, a tabu list is used to decide whether a city is visited or not. Moreover, 2 types of ants are proposed, namely *queen ants* and *worker ants*. *Queen ant* represents a thread block (Iteration level) where each thread represents a *worker ant* that can visit a city . In this proposed version, each thread (*worker ant*) loads a heuristic value $choiceinfo[i][j]$ which is proportional to the probability for an ant located in city i to visit the next city j (solution level). Then, a selection process is performed via Independent roulette wheel selection mechanism (*I-Roulette*), which is an alternative selection method. It generates a random number per city. Then, for each city, this random number is multiplied by the heuristic value, and by the tabu list value (this value is to represent whether the concerned city has been already visited or not). The result of these multiplications are stored in an array on which a reduction operation is performed to select the next city to visit.

Following the work proposed in [21], three parallel implementations of the Ant Colony System (ACS) have been proposed in [68]. The first implementation ACS-GPU adds elements to the partial solutions in parallel. If simultaneous memory read and write operations on the pheromone table are performed, then some of the pheromone values can be lost. As a result, ACS-GPU implements local pheromone update using atomic operations to behave as the sequential implementation. The second implementation, called ACS-GPU-ALT, is designed to achieve more speedup. Its solution construction phase and local pheromone update are parallelized. Since it does not the atomic operations, it will have an impact on the probability to select the next edge by ants. However, according to [68], it will not lead to the construction of invalid solutions. The third implementation ACS-GPU-SPM, which is the main contribution of this paper, replaces the pheromone matrix by a selective pheromone memory of smaller size. The idea behind the selective pheromone memory is to pick the most important edges, in terms of pheromone, and put them in a memory. Indeed, the authors have noticed that, during the search process, only a subset of edges are selected because they have a high amount of pheromone.

Unmanned Aerial Vehicles (UAV) path planning problem is close to the TSP. The aim is to plan a path for UAV to visit sensor nodes within a wireless sensor network keeping a minimum cost (decreasing the length of the path, fuel consumption and mission time). A parallel ACO based on GPU has been proposed in [22] to solve the problem. According to the authors, the algorithm has four main steps: the first three are computed once, and the fourth is repeated until a termination criterion is satisfied (tour construction). The first step generates N random numbers for N sensor nodes using cuRAND library. The second step calculates the distance table between the sensor nodes using a kernel of N blocks and N threads, then the initialization occurs by assigning each ant to a random point (sensor node). Inspired by the approach of [21], each ant is represented in [22] as a block running N threads. Each ant chooses to visit the next node (iteration level) according to an appropriate model using the random numbers generated earlier (solution level). When the ants finish constructing their tour,

they calculate their fitness values, and update the shared pheromone table. By sharing pheromone table, read and write operations are performed sequentially while updating the table. The algorithm has been tested on 6 instances from TSPLIB to demonstrate its performance. It shows that GPU based ACO is faster, and it gives more accurate results than the sequential implementation.

A GPU accelerated ACO has been proposed to solve vertex coloring problem [56]. The implementation is based on *Recursive Largest First* (RLF) [26]. Their ACO algorithm places randomly m ants (m blocks of n threads where n represent the next vertices that an ant can visit) at vertices to build solutions (tour construction), and each ant builds a stable set by visiting selected vertices (iteration level). To select the next vertex to visit, the ant follows the pheromone values where pheromone values of adjacent two vertices are always zero (solution level). The ant repeats the selection until no vertex is selected, and it colors the vertices of the stable set. Afterwards, it restarts visiting vertices to find another stable set from the non colored vertices. When all vertices are colored, the number of colors is a solution for the problem. Next, the pheromone update phase is performed by two steps which are the pheromone evaporation and the pheromone deposit. The evaporation has been considered to avoid falling in local optima, and it is performed by decreasing each pheromone value. The pheromone values are multiplied by a fixed constant factor between 0 and 1. Then, the pheromone deposit is performed using the tours obtained by the m ants according to an appropriate equation mentioned in the paper. The pheromone values are stored in a 2-dimensional array and they are updated using a kernel of m blocks composed of n threads. The threads of each block load the i_{th} row of the 2-dimensional array into the shared memory, in order to apply the evaporation and deposit operation in parallel (coalesced memory). The algorithm has been tested on 6 instances and has achieved a speedup of 36.81x.

Another version of ACO has been proposed in [35] to solve multidimensional knapsack problem (MKP), where the ants are divided into colonies. Moreover, only one ant from each colony performs the update of the pheromone table (the ant that has the best solution). To add an object to the knapsack, the ant computes the desirability of the n objects and chooses one among them. To perform this task, each ant (iteration level) runs n threads to compute the desirability (solution level). Afterwards, each desirability value is multiplied by a random number between 0 and 1. The object that has the largest value is picked. The update of the pheromone table is performed in a separate kernel of c blocks and n threads, where c represents the number of knapsacks and n represents the number of objects. The pheromone values are updated by multiplying the values of the pheromone table by 1-ev where ev is an evaporation factor. If $ev = 0$, then all pheromones evaporate and the algorithm becomes a generator of random solutions, but if $ev = 1$, there will be no pheromone evaporation. The pheromones are updated by a factor $1.0/(1.0 + Best - Roundbest)$, where *Best* is the fitness of the best solution known so far for the colony and *Roundbest* is the fitness of the best solution of the current round. This algorithm is not very effective in terms of quality because several algorithms as [11] and [15] outperform it, but it is faster compared to these algorithms.

In 2014, a parallel ACO has been presented for edge detection in images [29]. An image is represented in memory using a 2D array representation of a graph. At the construction tour phase, each ant moves to neighbor node (horizontally, vertically or diagonally), and updates the pheromone matrix. Each node constructs its own

solution using the transition rule to select the next neighbor node. If an ant has visited all the neighboring nodes, it moves randomly to another node in the graph, then the pheromone level of every node is updated according to an appropriate equation, and considering also an evaporation rate to avoid stagnation of the colony. Unlike [21], where ants are assigned to thread blocks, it has been found in [29] that it is a wasteful use of GPU to assign an entire thread block for one ant because most of threads will be idle at each iteration. According to [29], assigning four ants into a thread block (iteration level) using a total of 128 threads (solution level) and operating on a warp level yields a better performance. The algorithm has been executed entirely on GPU and has achieved a speedup of 150x faster than the sequential version.

ACO has been adopted in [32] to simulate the pedestrian movement, where an equal number of individuals (two equal groups of pedestrians placed in the opposite sides) try to reach the other side of an environment (2D grid divided into cells where the environment size is multiple of 16) following an optimal path. Each individual is surrounded by eight cells (except individuals who are placed at the top and the bottom cells). The individual selects directly the forward cell if it is available. Otherwise, it selects an available cell based on the transition rule of ACO. When the individuals finish the tour construction (reaching the opposite side), pheromone update stage is performed. In their parallel implementation, a kernel of n blocks and 256 threads (n * 256 is the size of the environment) is launched to calculate the availability of surrounding cells (iteration level). Then, their distances are used to compute the transition rule (solution level). The next kernel selects the next cell for the pedestrian according to the transition rule. The third kernel composed of n blocks and 256 threads is responsible for individuals movement. It updates also the matrices of pheromone and the state of environment. Finally, the last kernel initializes the values of the scan matrix (matrix used in the early stages to store the transition probability values) to zero.

Later in 2015, the Satisfiability Problem (SAT) has been solved in [88]. A SAT instance is represented as a matrix of clauses denoted by q of boolean values. To test a solution candidate d which is represented as an array, an operation $q_{ij} \wedge d_j$ bitwise for each literal j of each clause of the SAT instance is computed such that $1 \leq i \leq b$ and $1 \leq j \leq n$, and we say that d satisfies q if and only the resulting matrix from the last operation contains at least one TRUE for each clause of the instance.
Their proposed parallel ACO creates an artificial ant colony that looks for a solution candidate (*ChooseSolution* routine or tour construction). Then, it evaluates it (iteration level). *ChooseSolution* launches a kernel to compute the literal probability p_l and to generate a random number r_l for each literal (each thread represents a literal). The assignment of the literal is TRUE if $r_l \leq p_l$, and FALSE otherwise. For the solution evaluation (solution level), three kernels are launched:
The first kernel is launched with a 2-D grid of blocks of length n in x-dimension and b in y-dimension, each block size is 32*32 threads (each thread represents a literal). The second and the third kernels calculate the sum of the solved clauses and they evaluate the quality of the solutions respectively according to an appropriate function mentioned in [88].
The second stage updates pheromone levels using the best solution known by the ant colony. This process is launched by a kernel of 512 threads, where each thread performs evaporation and deposit processes. The last stage is called

$blurPheromones$. It adds a certain computed value $r_l.ph_l$ to each pheromone quantity where r_l is random number in [$-max_i$, $+max_i$]. max_i is computed as follows: $max_i = u.e^{i/\sigma}$ where u is the blurring base value and σ is a decline factor. Their results show that the parallel ACO implementation runs 21x faster compared to its sequential version.

Particle swarm optimization (PSO) for combinatorial problems:
Standard PSO has been adapted to solve the multidimensional knapsack problem (MKP) in [89]. Since PSO is not naturally appropriate for combinatorial problems, a sigmoid transformation function has been used, in order to adapt PSO update position equation to the MKP. In the parallel design of PSO, particles are initialized by a kernel involving random number generations. Afterwards, the main loop of the algorithm consists in the following steps. First, the particles are evaluated using a separate kernel that performs this operation by assigning a single block to each particle (iteration level). Within the kernel, each thread evaluates one item (solution level), it reads its item profit, and then multiplies it by 0 or 1 based on the existence of the item in the solution. When the results of all threads are collected, a reduction operation is performed. Then, a decision is made whether to replace the best local position or not. The best global position is computed then by performing again a reduction operation using the library. After defining the best global and local positions, updating particles is performed using another kernel, which is divided into p thread blocks and every block divided into d threads (p represents the number of particles within the swarm and d represents the number of dimensions), where each thread updates one dimension according to the PSO equation. The results show that the GPU implementation outperforms multi thread CPU version by a factor of 3.5 to 9.6, depending on the problem size.

In 2015, two parallel approaches which are GPU-PSO and GPU distributed PSO (GPU-DPSO) have been presented in [28] to solve max constraint satisfaction problems (Max-CSPs). The mathematical formulation adapted to define the movement of a particle in the search space is taken from [66]. The parallel design of GPU-PSO uses four kernels: the first kernel initializes the population (iteration level). The second kernel evaluates the fitness of each particle by calculating the number of constraints violated (iteration / solution level). The third kernel updates the best local positions. Finally, the last kernel calculates the velocities and updates the positions using PSO equations. In the second approach (GPU-DPSO), the idea is to partition the swarm into sub-swarms (algorithmic level), where each sub-swarm contains particles that violate the same number of constraints. In the exchange phase, particles are transferred to a proper sub-swarm based on their new fitness. If there is no appropriate sub-swarm for a concerned particle, a kernel is launched to create another sub-swarm. In this design, each sub-swarm is associated with one block and each particle represents a thread while in the first approach, each particle is represented by one block and each thread computes one dimension. The results reveal that GPU-DPSO achieves better results compared to GPU-PSO in terms of execution time.

Other evolutionary algorithms:
Other algorithms have achieved a significant success in solving combinatorial problems. We start with solving Golomb ruler problem. The problem objective is to find a set of marks at integer positions of an imaginary ruler, such that no two pairs of marks are the same distance apart. The number of marks on the ruler is its order, and the largest distance between two of its marks is its

length (distance between the last and the first element). The Golomb ruler is optimal if no shorter ruler of the same order exists [62]. An evolutionary algorithm (EA) has been employed in [62] in order to solve the problem. The GPU based implementation runs the crossover, the mutation, and the memetic part (random modified hill climbing (RMHC) and simulated annealing are used) in parallel. First, the population is initialized on the host. Then, it is sent to the GPU to perform the other operations. In the first kernel, each thread generates a random number between [1, SizeOfRepresentation] for the crossover, while in the second kernel, each thread copies genomes between crossed over representations. The third kernel evaluates the individuals after the changes. The mutation kernel is run on a number of blocks. Each block copies a solution in the shared memory (iteration level). Next, all the threads run mutation operator on the solution (solution level). Each thread generates two random numbers to apply mutation (two random numbers mentioned earlier). After the mutation process, a reduction operation will be performed over the blocks (based on solution fitness) in order to select the best mutated solution. Their GPGPU implementation shows that it is 10 times faster than an optimized multicore CPU.

Scatter Search (SS) is a population-based metaheuristic identified by the following components:

- Initialization(): a generation method of an initial set of solutions.
- Improvement(): a procedure to improve the quality of the solutions (e.g., Tabu Search).
- UpdateReferenceSet(): a procedure to generate a reference set of solutions using the initial solutions.
- GenerateSubset(): a procedure that selects solutions from the reference set, then arranging them in groups.
- SolutionCombination(): a method that creates new solution by combining information contained in a concerned group of solutions.
- RestartReferenceSet(): once the reference set is stagnating, a refresh method is used to generate new solutions.

According to [63], one of the suited components of SS to parallelism is the improvement method. To implement the improvement method, two algorithms are tested: random mutation hill climbing and simulated annealing. The improvement method is run on a kernel composed of several blocks (each block represents a solution). When random mutation hill climbing is used, the threads perform the mutation operator on the solution (solution level), and they exchange between each other the optimal solution by a temporary minimal ruler found in each block. When simulated annealing is used, each thread stores a single solution. The threads synchronize after a given number of iterations. In the other hand, a crossover operator is also implemented in the improvement method, and considered to be parallelized, but its impact appears only on the big instances. At the generate subset stage of SS, K-means algorithm is used for the clustering where two kernels are used. The first is responsible to classify solutions that belong to the centroids, and each thread is responsible for the classification of each solution (iteration level). The second kernel is responsible of the centroids update. Another phase is considered to be parallelized which is the initialization where GRASP is implemented in such a way that each thread is responsible for one solution. The authors noticed that parallelizing the improvement method has the greatest impact on performance compared to other components of SS. The results can be seen in Table 2.

Another algorithm, called the systolic neighborhood search (SNS) [60], has been used in [79] to solve three combinatorial optimization problems: massively multimodal deceptive problem (MMDP), subset sum problem (SSP), and maximum cut problem (MAXCUT). SNS is defined as a set of cells (GPU threads in our case) that create a mesh network between each other. Each thread creates a solution (iteration level), and performs some operations like mutation and crossover to improve it. Then, it sends it to the neighbor cell. A CPU-GPU cooperative hybrid implementation of an hybrid systolic search has been proposed in [79]. In this design, a CPU thread 1 is dedicated to transfer data to GPU. Then, a CPU thread 2 invokes the GPU kernel. In order to keep CPU busy, CPU thread 1 is run in parallel with SNS using uGA, in order to generate a set of new solutions. This set is inserted to SNS to look for a quality improvement. The paper sets few requirements for the algorithm that run on CPU which are: fast, light, and capable of interacting with GPU several times. The algorithm has achieved a speedup of 118x for the MMDP, 365x for the SSP, and 166x for the MAXCUT with an average of 216x for the three problems compared to the CPU sequential implementation.

Min-Min heuristic and Cellular Genetic Algorithms (CGA) have been presented in [64] to solve scheduling of independent tasks problem. Graphic cell is the proposed algorithm which is a new parallel design of the CGA. In Graphic cell, the population is initialized randomly except one individual that is generated using Min-Min heuristic. The algorithm uses two recombination operators that are specifically designed for algorithms implementing cellular topologies. The two proposed operators follow the same design: the offspring solution is generated by assigning to each task the machine of one of the neighboring solutions. These operators differ only in the criterion used for the selection of the neighbor solution to perform recombination:

– Fitness: the probability of selection of a neighbor solution is proportional to its fitness.
– The completion time: the probability of selection of a neighbor solution is proportional to the estimated time of one machine to perform a considered task (the machine is in the neighboring solution).

These selection mechanisms are both used each time with a probability *Psel* for fitness based operator and 1-*Psel* for completion time based operator. This decision is made for every task, and not for the entire solution. These recombination operators are implemented within two kernels. The first calculates the probability for each solution (iteration level) in the neighborhood to be selected. The second kernel randomly selects the recombination operator (UPRf or UPRct) to be applied where one thread is assigned to one task per solution (solution level). The algorithm implements three other kernels by assigning one thread per solution. The first kernel changes the assignment of a chosen task to a randomly chosen machine (mutation). The second kernel computes the fitness for the solution (*makespan*). Finally, the last kernel replaces the old solution with the new computed solution if it is better. We did not find clear results that show the performance of GPU implementation over CPU. Actually, the results shown in the paper are related to the effect of the two proposed recombination on solution quality, and the advantage of Graphic cell over the Min-Min heuristic.

Earlier in 2013, a parallel memetic algorithm has been proposed to solve

the task scheduling problem in heterogeneous environments [54]. An adaptive sorting strategy to achieve maximum speedup is proposed. This work presents a hybridization between a genetic algorithm and the variable neighborhood search (VNS). In their parallel algorithm, the population is initialized by randomly generating individuals. The initialization is performed on GPU using a kernel of *PopSize* x *Tasks* (iteration level/solution level). Each thread initializes one cell in a considered schedule by generating a random number (this random number represents the processor number to which a considered task is assigned). In the other hand, the evaluation is performed by a separate kernel of *PopSize* threads, where each thread computes the *makespan* of every solution. The selection, the crossover and the mutation are also parallelized in two kernels of *PopSize* threads (selection and crossover in one kernel). For the selection, each thread selects randomly a parent solution from the population to cross with its concerned solution. Then, the crossover is applied, and the mutation is run with a predefined probability (i.e., mutation rate). Finally, to perform the replacement strategy, the population must be sorted based on the makespans. For this sorting operation, an adaptive sorting strategy is proposed to choose the sorting mechanism, this latter is based on the population size. A parallel single thread block bitonic sort algorithm is selected when the population size is less than a specific threshold, while multiple thread block merge sort is chosen for the large populations. At this stage, parallel VNS is applied on all solutions in one kernel that is run with *PopSize* number of threads. This kernel calls four device functions which are shacking, local search, task priority, and evaluation. These functions are executed iteratively until the termination criterion is satisified. The algorithm could achieve a speedup even on the small populations by 696x compared to the CPU sequential implementation.
A CUDA framework implementation based genetic algorithm (GA) has been used to solve QAP in [55]. The algorithm initializes in parallel a population of *popesize* individuals by a kernel of *popesize* threads (iteration level), then it evaluates them. To perform the selection procedure, *Ncross* individuals are selected randomly. Then, the best individual is picked among them to be crossed with the concerned individual. Parallel crossover is performed in a separate kernel of *Ncross* threads using one point approach, where a crossover rate is considered to represent whether to perform the crossover or not. Afterwards, the mutation is run by performing a permutation between two random points in the individuals. To avoid falling in a local optimum, the replacement operation is performed. It selects the half of the new population to be in the next generation, while the other half is selected randomly. The experimentation has been performed on 10 instances of QAPLIB and it shows an acceleration factor of 30x compared to the sequential implementation.

Connected sensors that cover all discrete targets in a given network is called the connected sensor cover (CSC). An artificial bee colony algorithm (ABC) on GPU has been proposed in [65] for minimizing the number of connected sensors that cover all discrete targets (CSCDT) to reduce energy and communication cost. The algorithm constructs in CPU an initial solution CSCDT using *repeateddeletion* method which removes the sensors if they are redundant (area can be covered by another sensor). Then, the solution is copied to all m bees. Using a kernel of m threads, each bee (thread) executes *Sensorreduction* (iteration level) that removes two adjacent sensors s_i and s_j from the solution if a sensor s_k exists from the vertices set that does not belong to CSCDT and that covers the areas of both

Table 2.: Characteristics of GPU-accelerated metaheuristics on combinatorial problems

Ref	Problem	Algorithm	Parallelism level	Acceleration	Benchmark	Quality improvement	System CPU/GPU
[62]	Golomb ruler	EA	Iteration level + solution level	10x	8 instances of different length of ruler	Optimal solutions in three instances and deviation of 30% for the other instances	CPU: QEMU Intel 64-rhel6 (2.4 GHz)/ GPU: NVIDIA Tesla m2090 with 512 cores at 1.3 GHz
[63]	Golomb ruler	SS	Iteration level + solution level	1.22x	4 instances of different length of ruler	Quality improved	CPU: Intel Xeon E5645 (2.40 GHz)/ GPU: Nvidia Tesla m2090 with 512 cores at 1.3 GHz
[21]	TSP	ACO	Iteration level + solution level	20x	TSPLIB [7]	Quality improved	CPU: Intel Xeon E5620 Westmere processor (2.40 GHz)/ GPU: Nvidia Tesla C2050 Fermi graphics card endowed with 448 cores at 1.15 GHz
[68]	TSP	ACO	Iteration level + solution level	24.29x	TSPLIB [7]	Quality improved by ACS-GPU-SPM	CPU: Intel Xeon E5-2670 (2.6 GHz)/ GPU: Nvidia Kepler GK104 with 1536 CUDA cores at 745 MHz.
[22]	TSP	ACO	Iteration level + solution level	546.66x	TSPLIB [7]	Quality improved	CPU: Intel i5 Sandy Bridge (3100 MHz)/ GPU: NVIDIA GeForce GTX 680 Kepler with 1536 CUDA cores at 1002 MHz
[32]	The pedestrian movement	ACO	Iteration level + solution level	18x	2D grid and agents manually generated	No improvement	CPU: Intel Core i7-930 (2.8 GHz)/ GPU: Nvidia GeForce GTX 560ti with 448 cores at 1.464 GHz
[56]	Vertex coloring	ACO	Iteration level + solution level	19.68 to 36.81x	6 instances from [2]	Quality improved for one instance	CPU: Intel Core i7-4790 (3.66GHz) /GPU: NVIDIA GeForce GTX 1080 with 2560 cores at 1544 MHz
[35]	MKP	ACO	Iteration level + solution level	575x	ORLIB [12]	No improvement	Not mentioned
[29]	Edge detection	ACO	Iteration level	150x	The standard test images [6]	Quality improved compared to [57], Sobel and Canny algorithm	CPU: Intel i7 950 (3.06 GHz)/GPU: NVIDIA GTX 580 with 580 cores at 1544 MHz
[88]	SAT	ACO	Iteration level + solution level	21x	Benchmark instances from the SAT CNFs	No improvement	CPU: Intel core i7 3770K (4.5 GHz) /GPU: NVIDIA Geforce GTX 680 with 1536 processing cores at 1.058 GHz
[79]	MMDP, SSP, MAXCUT	SNS+uGA	Iteration level	216x	Large instances with k = 20, 30, 40 for MMDP, instances generated as described in [45] for SSP, MAXCUT instances generated as described in [45] and [3]	Quality improved	CPU: Intel i7 920 (2.66 GHz)/ GPU: GeForce GTX 650 with 384 cores at 1058 MHz
[64]	Task scheduling	Graphic cell	Iteration level+Solution level	Not mentioned	Instances created as described in [10]	Quality improved	CPU: Intel Xeon E5440 (2.83 GHz)/ GPU: Nvidia Tesla C2050 with 448 cores at 1.15 GHz
[54]	Task scheduling with precedence relations	Memetic algorithm	Iteration level + Algorithmic level	696x	Instances chosen from benchmark of [84]	No improvement	CPU: Intel Xeon processor (2.8 GHz)/ GPU: GTX480 with 480 cores at 1006 MHz
[85]	Probabilistic TSP with deadlines	RRLS	Iteration level + solution level	10x	Benchmark instances from [20]	Quality improved	CPU: Quad-Core AMD Opteron (2 GHz)/GPU: GeForce GTX 580 with 512 cores at 1544 MHz
[55]	QAP	GA	Iteration level	30x	10 instances from QAPLIB [5]	Quality improved in 5 instances	Not mentioned
[89]	MKP	PSO	Iteration level + solution level	3.5 to 9.6x	Benchmark taken from [23] and [83]	No improvement	CPU: quad core intel i7-920 (2.66 GHz)/ GPU: NVIDIA GTX 580 with 512 cores at 1.5 GHz
[28]	Max-CSPs	2 variants of PSO	Iteration level + solution level + algorithmic level	1.318x compared to GPU-PSO	Randomly generated instances using [69]	No improvement	CPU: Intel core i7-2630QM (2.0Ghz)/ GPU: NVIDIA GeForce GT 525M with 96 cores at 1200 MHz
[65]	Small number of connected cover sensors	ABC	Iteration level	5x	100x100 square area, sensors and targets are randomly located in the region, the number of targets is 20. Sensing radius and communication radius of each sensor are both 10	No improvement	CPU: Intel Core i3-4130 (3.4GHz)/ GPU: NVIDIA GeForce GTX 760 with 1152 cores at 980 MHz

s_i and s_j. Then, each bee evaluates the new solution according to the number of deleted sensors by *Sensorreduction*. Afterwards, the bee becomes a *recruiter* or a *follower* by a given probability. If the bee B_j is recruiter then it maintains its $CSCDT_j$. Otherwise, it selects one of CSCDTs owned by recruiters according to

a certain probability defined by an appropriate formula. In this design, sensors data and targets are copied in constant memory, variables of the kernel function are stored in a register, and all informations are exchanged via the global memory. The experimental results show that this algorithm is 5 times faster than the sequential version.

According to [85], the evaluation of the objective function and constraints are the most expensive tasks. Since a lot of metaheuristics generate a set of solutions to be evaluated, the simplest approach is to evaluate the solutions in GPU. Unfortunately, this approach leads to two main drawbacks: the first is that the number of solutions has to be sufficiently large to achieve an efficient use of GPU. The second is that the evaluation does not allow for straight flow without branches and jumps (thread divergence). To overcome these drawbacks, a novel framework is proposed in [85]. The framework is designed to problems for which the objective function and constraints can be evaluated approximately using Monte Carlo sampling. The idea is to create a set of samples according to a given probability distribution. These samples are used then for an approximate evaluation. As an advantage, it is possible to parallelize the evaluation of one sample instead of one solution. It leads to a much better grade of parallelism. Besides, the evaluation of one sample shows a simpler flow control with less branches and jumps. To test this framework, a parallel random restart local search has been applied on the probabilistic traveling salesman problem with deadlines achieving a speedup of 10x compared to CPU sequential implementation.

3.2. Solving continuous problems by GPU accelerated metaheuristics

Thanks to GPUs, tackling high dimensional continuous problems has become also relatively straightforward. By exploiting GPU, fitness evaluation, computation, and convergence to near optimal solutions are handled efficiently. In this section, we outline a set of papers addressed to solve continuous optimization problems. We conclude it with Table 3 that presents the characteristics of the implementations and the results.

Particle swarm optimization (PSO) for continuous problems:

In 2016, a parallel implementation of PSO using GPU has been presented in [42]. Their implementation has been tested on a benchmark of known optimization functions. Besides, it has achieved a speedup of 46 times faster than the sequential algorithm. Their GPU proposition consists in seven kernels with a ring topology to form a virtual neighborhood for particles. The first kernel allocates memory on GPU, and each thread uses *Curand* library to generate random numbers needed for the algorithm. The second kernel initializes basic information, such as position and velocity. The third kernel generates ((n+32-1)/32) blocks of 32 threads for computing the fitness function (Solution level) where n is the number of individuals in the swarm. It performs a reduction process to calculate the fitness values. Then, via these values, *Pbest* is updated. The fourth kernel is responsible of calculating local best *Lbest*. In this kernel, each particle compares his *Pbest* with its neighbors *Pbest* (left and right neighbors). The fifth kernel computes the velocities and positions for the next iteration(solution level/iteration level). A sixth kernel computes the global best position of the entire swarm using atomic functions. The last kernel liberates the structures used on GPU. The results have shown the positive impact of coalesced memory on the acceleration. However, the speedup is decreased when the population size is more than 2000 and dimension size is more than 50.

The traditional PSO algorithm has been improved in [50], where a new parallel algorithm that employes two levels of PSO has been proposed. In the bottom level, the particles are divided into N groups, each of which runs the PSO (algorithmic level), and sends the best particle to the top level individually. Afterwards, the bottom level updates its own particles (position and velocity) according to the top level results. In their parallel implementation, the method is designed as each thread represents a particle, and each block represents a group of particles (iteration level). The algorithm has six main steps: initializing all the particles in the CPU side, updating the speed and the velocity of the particles at the bottom level of PSO, generating top level particles by gathering the best particles from the bottom level, executing PSO by the top level particles, mixing the top level and the bottom level particles together, and finally sending the best particle of one group to its neighbor group. The results have shown that it can achieve good convergence and speed.

An implementation of parallel cooperative PSO has been presented in [49]. In this version, the population is divided into different subpopulations. Each subpopulation optimizes only one component. Their GPU implementation consists in six kernels. The first initializes the position and the velocity of particles on GPU instead of CPU. The second is responsible of evaluating the fitness of all the particles (solution level/iteration level). In this kernel, one block of threads was mapped to evaluate the fitness of one individual. Next, another kernel is run, where one thread block is used to find the global best position. A separate kernel is used to update the position of each particle. Then, another kernel is invoked with one block of threads to update the global best position. Finally, position velocity update Kernel is run, where one thread is updates one element from velocity and one element from position vectors using the appropriate equations of PSO. The algorithm has been tested on Shifted Sphere, Shifted Elliptic, Shifted Rastrigin, Shifted Rosenbrock, and on Shifted Ackley and the results have shown a great reduction of time.

An alternative version of PSO has been implemented in [44] along with a communication strategy, called genetic migration ($PPSO_{GM}$). It adopts coarse-grained parallel model, and uses N subswarms. Each subswarm runs the classical PSO independently (algorithmic level). Considering that each subswarm is represented as *individual* of a genetic algorithm, and the members within the subswarm are considered as variables, the communication between subswarms is established by implementing genetic operators (selection, crossover, mutation) on them. Besides, the fitness of each individual is the best fitness of variables within the subswarm. In their parallel implementation, each block represents a subswarm, and the threads represent particles (iteration level). The threads calculate the fitness, and update the velocity and the position in parallel. The algorithm uses a set of kernels. The first kernel initializes the subswarms, and makes the particles evolve for a certain number of iterations. Next, another kernel implements the genetic migration strategy, in order to exchange individuals between subswarms. The algorithm is terminated after a certain number of communications. According to the authors, $PPSO_{GM}$ has a better convergence consuming approximately the same time as PPSO with unidirectional ring migration. Besides, it has achieved a global speedup of 56x when using subswarms of 1024 particles and a speedup of 300x when using subswarms of 16384 particles compared to the CPU sequential implementation.

Achieving an average speedup of 17x, Dynamic Cooperative Hybrid MPSO+GA

has been designed in [36]. MPSO+GA alternates between Multi-Swarm PSO (MPSO) and genetic algorithm (GA). The proposed design integrates the following genetic operations: mutation, crossover, and selection into the update phase of PSO. Besides, particles are divided into subswarms forming a ring topology (algorithmic level). GA operates by applying the same three operators to the population. GA uses tournament selection to pick groups of individuals. Then, the best individual is selected within each group for crossover (each group executes in parallel). In the mutation phase, genes in the individual are mutated with defined probability between 0 and 1. Moreover, a heuristic is used to change the algorithm if it does not improve the best solution found during a certain number of iterations. The parallel version of this design is expressed in 8 kernels and they are implemented as follows:

(1) Particle Initialization Kernel: using one thread per particle dimension (solution level), this kernel initializes the positions and the velocities vectors (iteration level), i.e. position is randomly initialized between $[-x_{max}, +x_{max}]$ and velocity is initialized to zero.
(2) Update Fitness Kernel: in this kernel, multiple threads cooperate to compute the fitness of each particle. First, each thread reads four position values from global memory, computes the values of the corresponding four terms to perform a partial sum for them. Next, all the partial results are used to perform reduction, in order to compute the fitness value of one particle. Finally, the final fitness of all particles are written back to the global memory fitness buffer.
(3) Update Bests Kernel: it uses one thread per particle. First, each thread compares the new and old local fitness for its particle in order to update the best local position. Next, the kernel performs a reduction operation to update the best global position for each swarm.
(4) Update Position/Velocity: it uses one thread per dimension. Velocity and position are computed using the equations of PSO. Afterwards, mutation is performed on both position and velocity with a defined probability β. 5-Find Best/Worst Particles: this kernel performs a reduction operation to find the best and the worst particles in each subswarm.
(5) Swap Particles Kernel: this kernel performs the exchange between subswarms. Each thread represents one dimension to be exchanged, and a given number of the best particles in subswarm j overwrite the worst particles in subswarm $j+1$ mod s (ring topology).
(6) Mutation Restoration Kernel: this kernel is proposed to recover the fitness, velocities and positions of *unhealthy* subswarms if the mutation performed on the previous iteration has led to very bad results. In this kernel, each thread restores 4 dimensions of each particle after checking if the subswarm is *unhealthy*. To do so, it compares the current fitness with the last fitness that was before mutation. If it is worse, old position and velocity from their saved buffers are restored.
(7) Crossover and Mutation Kernel: for each particle, t/4 threads randomly select 4 particles, and perform a parallel reduction in order to find the best fitness. With a probability γ, a single point crossover is performed by randomly choosing a shared crossover point between all the threads. Besides, half of the threads reads the right half of the first parent, and the other half reads the left half of the second parent. Afterwards, one thread combines these chunks. In the same kernel, mutation is applied by all the threads to their chunks of values.

In 2015, a parallel PSO has been designed in [52] using CUDA. This design is handled in a collection of remote computing services, called Amazon Web Services Cloud (AWS). According to the authors, the calculation stress of PSO is proportional to the size of the particle. It means that the larger the particle is, the greater the pressure is. The following phases are parallelized:

– Calculation of the fitness values of particles: one particle is represented by one block and each thread computes one of its dimensions (solution level/iteration level). Then, partial results are reduced to thread 0 that writes the final fitness value in the global memory.
– Update best local position and best global position: since updating positions needs to update each dimension of each particle, each particle is represented by one block, and each thread updates each dimension of the particle. Finally, the last kernel updates position and velocity of each particle.

Their experiment has been performed by comparing CPU implementation with GPU local implementation, and GPU AWS implementation. Their results show that the GPU AWS based PSO runs 80 times faster than the sequential algorithm, and 64 times faster compared to GPU local implementation.

Differential Evolution (DE): Earlier in 2012, a parallel DE algorithm has been presented in [90], combined with an elite opposition-based learning strategy (EOBL). The proposed parallel EOBDE consists in two parts which are DE and EOBL strategy [76]. The algorithm initializes a population randomly (iteration level). Then, EOBL is applied. The strategy firstly selects the best 20% individuals of the population as a set of elite individuals. Next, the dynamic interval boundaries are updated according to an equation mentioned in [90]. Then, *opposite* solutions are generated according to a model mentioned in [90]. Finally, the fittest individuals are selected to be in the population. The algorithm handles these operations by implementing three kernels: the first kernel finds maximum and minimum values of each dimension. The second kernel generates opposite solutions of the elite individuals. The last one is designed selects the fittest individuals to insert them into the next population. To perform this task in parallel, 2 x NP individuals are assigned to each thread block (iteration level). Each individual is compared with other 2 x NP -1 individuals to calculate its rank value in order to find the members of the new population. Afterwards, DE is applied. To handle DE operators (mutation, crossover and selection), individuals are represented by threads within a separate kernel. As mentioned above, This Algorithm has been tested on 10 functions with dimensions 500 and 1000. Besides, it has been compared with 4 other algorithms from the literature. The results show that EOBDE achieves the best results on 8 of the 10 problems when D=500. In case of D=1000, EOBDE performs better for 9 functions. Besides, EOBDE shows an average speedup of 4.475x compared to the sequential implementation.

A hybridization between DE and Backtracking Search Optimization Algorithm (BSA) and Simulated Annealing (SA) has been carried out in [16]. The algorithm consists mainly in two stages. The first stage consists in five phases. The first, called Selection-I is a backtracking strategy to store the old population of the previous generation (history). This populations is replaced with a probability of 0.5 by the current population. Afterwards, the mutation of individuals phase takes place, where a hybrid equation is proposed by combining mutation equations of both BSA and DE/target-to-best/1 where a SA schedule is proposed to decrease the scaling factor. Next, two crossover strategies are randomly used

(with probability 0.5) to generate a new trial population T from the current and the mutant population. The first strategy depends on a parameter that controls how many dimensions of the mutant will be incorporated in the trial individual. The second strategy ensures that only one dimension from the mutant individual will be concerned in the new trial individual. Due to the fact that the mutation equation can generate trial individuals outside the search space, the confinement phase has a role of regenerating the dimensions that are out of the space inside the appropriate bounds. Finally, in Selection-II phase, T_i replaces an individual P_i if it is better. The second stage performs a DE/target-to-best/1 iteration on the worst individual. In their GPU based implementation, for the most of the phases (for example, mutation, evaluation) the algorithm assigns to each individual a block (iteration level), and to each dimension a thread (solution level) to compute it (kernel of N blocks of D threads). However, sometimes another data decomposition is needed: for example, a thread is assigned to each individual in order to update the global best solution.

According to [78], detecting objects in images is a frequently tackled problem in computer vision and pattern recognition. This problem can be turned into a continuous optimization problem, as it is done in [78]. A parallel PSO and DE have been proposed to tackle two problems in this field: hippocampus localization in histological images and human body pose estimation in video sequences. The objective of human body pose estimation in video sequences is to estimate accurately the posture of human body in a video stream. In this problem, the input is N views of the body from several angles. Afterwards, the silhouette of the body within each image is extracted. The silhouette is a binary image where all pixels belonging to the body are set to 1. To solve the problem, three steps are followed. First, a pose estimation is generated by the search algorithm using an appropriate parametric model. Then, a 3D rendering of the body is applied for the pose. Finally, a set of N images, corresponding to the projections of the rendered body (silhouettes) on the image planes of the input is computed. For further details about the parametric model used, we refer the reader to [78].
In CUDA-based implementations of PSO and DE, three kernels are implemented. For PSO, the first kernel initializes and updates the velocity and position of all particles. The second kernel evaluates the fitness. Finally, the third kernel updates the best positions. In DE, the first kernel generates the offspring solutions. The second kernel evaluates the fitness of all produced solutions. Then, the third kernel performs the selection of the new population. PSO and DE have the same structure, each thread block is responsible of one particle (iteration level), where each thread updates one dimension of the problem (solution level). The experimentation shows that PSO gives more accurate results than DE when dealing with human body pose estimation. However, DE gives slightly better results in case of hippocampus localization in histological images without mentioning any details about the speedup gained by GPU implementation over CPU.

Other evolutionary algorithms: A new design of a memetic algorithm, called MA-SW-Chains is presented in [51] for GPU architecture. Its main idea is to combine a steady-state genetic algorithm (SSGA) [86] with a local search procedure, called *Solis Wets* search method [70]. According to [51], the GPU accelerated MA-SW chains has two natural sources of parallelism: the number of individuals in the population (iteration level), and the number of variables of each individual (solution level). The steps that have been parallelized in the algorithm are: evaluation of the fitness function, adaptation of the crossover operator to the GPU, optimization

of the local search, random number generation process, and population sorting.

(1) Fitness function evaluation: each thread block is assigned the processing of a set of dimensions of each individual (iteration level). Within each block, each thread processes several variables (bucket). Each bucket processed is composed of interleaved elements to make thread warps access consecutive elements (coalesced memory). Afterwards, a reduction operation is performed to compute the partial result for each thread. The final fitness function of an individual is then computed by another reduction operation.
(2) Crossover: each thread processes a bucket of pairs of dimensions (solution level). Then, it writes the result in the memory. In fact, the crossover operation is designed such that each thread crosses $BucketSize$ elements of the two parent individuals. It means that each block generates $ThreadsPerBlock * BucketSize$ elements for the new individual. In the proposed design, n crossing operations are performed in parallel to increase the thread parallelism where n is the number of individuals. Another operation has been parallelized which is the Euclidean distance to select the parents to be crossed.
(3) Local search: the operations that have been parallelized are: individual change operations, bias values increment and decrement operations, individual substitution in the population, and fitness evaluation.
(4) The generation of random numbers and population sorting.

A GA has been parallelized in [67] to tackle continuous functions. In their proposition, the selection procedure is implemented using Roulette wheel selection function with a separate kernel, and it is performed by generating random numbers between 0 and the sum of the fitness values of the population. If the fitness of the corresponding individual is greater than the random number, then it becomes a parent chromosome. Afterwards, another kernel performs a uniform distribution crossover with a fixed ratio. Unlike single and double point crossover where mixing is done at segment level, uniform distribution crossover creates child chromosome at gene level (solution level). The authors claims that it is more suitable for the large populations. The last genetic operator is mutation. It is implemented in single kernel, where each individual is mutated by a thread (iteration level). The parallel GA has been tested on seven test functions and it shows to be faster with 4.15x than the sequential version.

A parallel bee algorithm (CUBA) has been proposed in [53] to tackle continuous problems. CUBA is a multi-colony bee algorithm that brings a good efficiency and a high speedup. The algorithm initializes the population and evaluates the fitness of individuals through parallel threads (iteration level/solution level). To get the best sites, the population needs to be sorted. For this reason, the authors have used Odd–Even Sorting algorithm. In their design, bees are grouped into colonies. Each thread is assigned to its colony according to the thread ID. In the standard bee algorithm, more bees are recruited for the best sites. However, in this proposition, the authors aims to balance the loading among the threads. They have proposed to assign *nep* bees to recruit m sites. The algorithm overcomes the overhead due to the communication between the colonies by using shared memory and adapting 2 phase communication strategy. CUBA has been applied on 9 minimization functions, and has achieved a speedup of 13x times compared to the standard sequential bee algorithm.

A parallel multi-objective tabu search (MOTS2) has been designed in [77] to

handle multi-objective high-dimensional problems, where the GPU acts as a co-processor that supports CPU by evaluating large number of solutions. The high level and complex parts of the code are performed by the host. However, the neighborhood evaluations are carried out in batches by the device (iteration level), where each neighboring solution will be evaluated asynchronously. As a result, the number of solutions to be evaluated determines the grade of parallelism. In other words, parameters are configured automatically within the kernel. The implemented algorithm has been tested on the ZDT and ZDT2 functions, and it has achieved an average speedup of 23.7x compared to the CPU implementation.

4. Discussion

This section is devoted to discuss parallel works that have been reviewed. It is based on the acceleration factor and the achieved solutions quality to demonstrate GPU role within metaheuristics.

Table 3.: Characteristics of GPU-accelerated metaheuristics on continuous problems

Ref	Algorithm	Parallelism level	Acceleration	Benchmark	Quality improvement	System CPU/GPU
[90]	DE	Iteration level	4.475x	Functions from CEC 2008 [75], and [40]	Results improved compared to CHC, DE, SOUPDE and GODE	CPU: Intel Core 2 Quad Q8200 (2.33GHz)/GPU: NVIDIA GeForce GTX 285 with 240 cores at 1476MHz
[16]	DE-BSA-SA	Iteration level + Solution level	40x	Functions taken from [24]	No improvement	CPU:Intel Core processor i5-3330 (3.00GHz)/GPU: NVIDIA GeForce GTX680 with 1536 cores at 1058 MHz
[78]	DE+PSO	Iteration level + Solution level	Not mentioned	4 test sequences made by the CVSSP, University of Surrey for Human body pose estimation + 15 images of hippocampi by manually segmenting the anatomical structures for Hippocampus localization in histological images	PSO performs better in human body pose estimation in video sequences but DE is better in hippocampus localization in histological images	CPU: Intel Core i7 CPU (2.80 GHz)/GPU: NVIDIA GeForce GTS450 with 192 cores at 1566 MHz
[51]	MA-SW chains	Iteration level + Solution level	82.17x	CEC 2010 [74] + a benchmark setup mentioned in [51]	No improvement	CPU: Intel Core i7-930 (2.8 GHz)/GPU: Nvidia Titan with 2688 CUDA cores at 837 MHz
[42]	PSO	Iteration level + Solution level	46x	Sphere, Rosenbrock, Rastrigin, Griewank, Ackley, De Jong, Easom	Quality improved in Sphere and Griewank function compared to the CPU sequential implementation	CPU: Intel Core i7 (860 MHz) /GPU: GeForce GTX 980 with 2048 cores at 1126 MHz
[50]	PSO	Algorithmic level+Iteration level	4.9x compared to HBPSO algorithm	Griewank, Rastrigin, Rosenbrock	No improvement	CPU: Intel Pentium G2020 (2.9 GHz)/ GPU: NVIDIA GeForce GT 630 with 902 cores at 1804 MHz
[67]	GA	Iteration level	1.18 to 4.15x	Function taken from [43]	No improvement	CPU: Intel Core i5 4200 (2.6 GHz)/ GPU: nVIDIA GeForce GT 740M with 384 cores at 810 - 980 MHz
[49]	PSO	Iteration level + Solution level	7.26x	CEC 2008 [75] and CEC 2010 [74]	Quality improved in Shifted Rosenbrock compared to the CPU sequential implementation	CPU: not mentioned/ GPU: Tesla M-2070 with 448 cores at 1.15 GHz
[44]	PSO	Algorithmic level+Iteration level	178x compared to $PPSO_{URM}$	Functions taken from [66]	Quality improved in all the functions compared to $PPSO_{URM}$ and the CPU sequential implementation	CPU: Intel Core i5-4670 (3.4 GHz)/ GPU: NVIDIA GTX660 with 960 cores at 980 MHz
[36]	PSO	Algorithmic level+Iteration level + Solution level	17x	CEC 2010 [74]	Quality improved compared to Static MPSO+GA and CPU sequential implementation but it doesn't achieve the best results of MPSO-MCS	Not mentioned
[53]	CUBA	Iteration level + Solution level	13x	Functions from [61]	No improvement	CPU: AMD Athlon II (3.0 GHz)/ GPU: GeForce GTX 460 with 336 CUDA cores at 1350 MHz
[52]	PSO	Iteration level + Solution level	80x	Sphere, Rastrigin, Griewank, Rosenbrock	No improvement	CPU: i5-4670 (3.4GHz)/GPU: NVIDIA GTX660 with 960 cores at 980 MHz
[77]	MOTS2	Iteration level	23.7x	ZDT and ZDT2 functions	No improvement	CPU: not mentioned/ GPU: Quadro 1000M with 96 cores at 1400 MHz

We have tried to cover as possible the recent works published from 2012. For the best of our knowledge, only few papers adapt single solution based metaheuristics to tackle continuous problems with GPU, like [77] where ZDT and ZDT2 functions have been addressed using multi-objective tabu search.

To adapt single solution based metaheuristics for GPU, iteration level is implemented to speedup the time-consuming generation and evaluation of the neighbors without affecting the behavior of the algorithm. Besides, algorithmic level is implemented for two reasons, which are to achieve a high occupancy of GPU, and to enhance the exploration ability of the algorithms by launching multiple instances [8, 18, 30]. We noticed that communication strategies are implemented between processes in order to guide the search towards promising search regions. For instance, in [8], processes form a ring topology and exchange solutions based on a diversification strategy mentioned in [19]. In [18], a working set F is used to exchange solutions. Solutions are exchanged if the last read solution has not been improved for a certain number of iterations, or if an improvement of the last read solution occurred. Compared to CPU sequential implementation, we noticed improved results thanks to the large generation of neighborhood, and to the communication strategies. Nevertheless, Table 1 shows that the acceleration factor of the described algorithms does not exceed 14.86x [82]. We think that it is a humble factor if we consider the huge computing power of GPU.

When using population-based metaheuristics to solve combinatorial problems, iteration level is always used regarding the large set of individuals being evaluated and evolved [21, 22, 28, 29, 32, 35, 54–56, 62–65, 68, 79, 85, 88, 89]. However, algorithmic level is considered only in few papers like [28], where partitioning the swarm into sub swarms is proposed. Each subswarm runs PSO separately. To further accelerate the execution time, the solution level has been also implemented to generate, evaluate solutions, or both in parallel [21, 22, 28, 29, 32, 35, 54, 56, 62, 64, 88, 89]. As an example, in papers like [21] [22] [32], ACO algorithm have been implemented, where tour construction phase can be performed in parallel. More specifically, an ant generates a set of possible moves in parallel. Then, it selects the best move. On the other hand, solution quality has been considered in works like [63, 64] with a small acceleration factor of at best 1.19x [63]. While in works like [32, 35, 54, 88], the behavior of GPU parallel implementations is not modified compared to the CPU implementations (quality not improved) but a significant acceleration factor of at best 696x [54] is achieved. Few works as [21, 56, 60, 62, 68, 81] made the exception. They improve the quality along with keeping a high acceleration factor.

In the same context, population based metaheuristics have been widely exploited to solve high dimensional continuous problems. Besides iteration level, in most of works, as [16, 36, 42, 49, 51–53], dimensions of individuals are computed in parallel (solution level), which increases the acceleration of the algorithms. Quality has been preserved, and even improved as it is the case in [36, 42, 44, 49, 90]. However, for the best of our knowledge, algorithmic level is less covered and found only in [36, 44, 50]. This level is implemented by partitioning a given population into subpopulations where a defined algorithm is run for a certain number of iterations. Then, a communication strategy is performed between them to exchange individuals.

Finally, the common point of the works in designing a GPU based algorithm is trying to maximize data parallelism. Works like [16, 21, 42, 50, 51] represent solutions as thread blocks where threads evaluate dimensions, or perform specific operators, such as crossover, mutation or to generate the neighborhood. Moreover, GPU impact becomes clear when big instances are addressed. We take as an example [71], in which solving an instance of 10 takes 0,33 s in CPU sequential implementation

and 2.12 s on GPU. However, solving an instance of 500 takes 4331,783 s in CPU sequential implementation, but it takes only 1628 s on GPU. Another example is taken from [55] solving QAP, where the CPU sequential implementation converges to the final solution in 1442.38 s for an instance of 100 while it takes 45.78 s using GPU.

5. Guidelines and conclusion

Inspired by [17] and from what we have reviewed, we cite some guidelines to build an efficient parallel GPU based metaheuristic. These guidelines are summarized as follows:

- To guarantee a GPU based metaheuristic performance, a relatively high occupancy of GPU is mandatory. It would cover the high latency of the memory. A high occupancy can be ensured by:
  (1) Increasing the size of neighborhood (single solution based metaheuristics).
  (2) Representing solutions as thread blocks, where threads are responsible of computing.
  (3) Increasing the number of individuals within the population (case of population based metaheuristics).

  However, very high occupancy can also degrade the performance [29]. The reason is the memory restrictions, e.g. registers memory is limited which means that each thread has a specific amount of registers. Increasing the number of threads would require using the global memory to overcome registers memory limitation [17]. As an example, we can cite [42], where a high speedup has been achieved, but the performance decreases when the population size is greater than 2000 and the problem dimension is greater than 50.
- Thread divergence: thread divergence can be handled by different mechanisms. These mechanisms are strongly related to the problem being solved and to the algorithm itself. From our review, one of the mechanisms that has been adopted to avoid thread divergence is to run many thread blocks that have few number of threads [54]; propose a tabu list to check whether a city is visited or not to solve TSP [21]; filter infeasible moves [18]; evaluate the individuals using Monte Carlo sampling to provide a simple flow control without branching during the evaluation [85].
- Coalescing memory access: ensuring coalescing memory access has a positive impact on the performance [1]. Papers cited in our review have shown the importance of coalescing memory access on the speedup of their algorithms, such as [27, 42, 51].
- When transferring data between the GPU and the CPU over the PCI express bus, it is recommended to use the so called Page Locked Memory. It will disable the memory paging (physical RAM continuous memory is guaranteed) [17].
- A further optimization is to use Write Combining Allocation. It disables CPU caching of a memory that the CPU will only write to, and improves transfer performance by 40% [17].

The main bottlenecks that can occur in the GPU based applications are: instruction throughput, memory throughput, latencies, or CPU-GPU communications. To locate kernel bottlenecks, several approaches can be used as employing a CUDA profiler [17]. One of the options of the CUDA profiler to locate kernel bottlenecks is to modify the source code and compare the execution time of the different kernels [17]. The profiler can identify whether a kernel is limited by bandwidth or by the arithmetic operations. This is done by using a strategy that adopts three modified versions of kernels: the original kernel, the math version and the memory version. In the math kernel, all memory load and store operations are removed. While in

the memory kernel, all the arithmetic operations are removed. Then, the runtime between the three versions can be compared [17].
In the same context, reducing data traffic to off-chip memory can be achieved through mechanisms such as kernel fusion. This recent strategy relies on the com -bination of kernels that share data arrays to larger kernels [34] where the shared data arrays are held by on-chip cache. Furthermore, a scalable method is proposed in [80] to find the best kernel fusion possibility. Kernel fusion is defined as an optimization problem where the optimal kernel fusion (in terms of performance) is found using a genetic algorithm. Besides, the understanding of memory access patterns and the efficiency of the different GPU storage types is essential for improving GPU based-applications. According to [41], finding the best way to make use of GPU memory (buffering strategy) is a challenging task. A buffering strategy mainly concerns the assignment control of the available storage types to given grid functions (stencil computations on arrays). Due to the multiple storage types and the grid functions, it is difficult to find the best buffering strategy. For given β buffering strategies and N grid functions, there are N^{β} possible configurations, which is a huge search space. Especially, if there are dozens of grid functions. This issue has been addressed in [41], where an assignment algorithm is proposed to find an optimal configuration along with a reduced search space to $O(\beta N^2)$. Moreover, a performance model is presented to measure the effects of the different storage types on several buffering strategies.
For further details about the kernel fusion or the buffering strategies, we refer the reader to [80] and [41] respectively.
Finally, following these guidelines and adapting an appropriate GPU implementation may lead applications (metaheuristics in our case) to a better performance.
As a conclusion, this survey shows that GPUs can play an important role in the efficiency of parallel metaheuristics for solving high dimensional problems. On the one hand, it can lead to high runtime accelerations and, on the other hand, solution quality is usually preserved and can sometimes be improved.
Although GPU based systems are used to speedup metaheuristics, it is still a bit early to say that GPU architectures are fully employed for metaheuristics, because the current strategies are strongly dependent to the algorithm or the addressed problem. Thus, the focus may be oriented towards new parallelization strategies that have the possibility to be reused from one problem or one algorithm to another. Besides, the design of new parallel metaheuristics will certainly be of high interest, especially if applied on hard real-world optimization problems, rather than on benchmark ones. It will also benefit from the convergence of several technologies (GPU, distributed algorithms, cloud computing, etc) and it will certainly give rise to interesting perspectives, in terms of new paradigms and applications.

References

[1] Global memory in CUDA. https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/. Accessed: 2017-04-02.
[2] Graph coloring problem instances. https://turing.cs.hbg.psu.edu/txn131/graphcoloring.html. Accessed: 2017-04-03.
[3] Maxcut problem instances. http://www.optsicom.es/maxcut/. Accessed: 2017-03-31.
[4] PSPLIB. http://www.om-db.wi.tum.de/psplib/. Accessed: 2017-03-31.
[5] QAPLIB. http://anjos.mgi.polymtl.ca/qaplib/. Accessed: 2017-03-31.
[6] Standard test images. http://sipi.usc.edu/database/. Accessed: 2017-03-31.
[7] TSPLIB. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html. Accessed: 2017-03-31.
[8] Omar Abdelkafi, Lhassane Idoumghar, Julien Lepagnot, and Mathieu

Brévilliers. A GPU-based parallel neighborhood evaluation for ITSSD. In *Proceedings of the 12th Biennal International Conference on Artificial Evolution, Lyon, France*, pages 327–334, 2015.

[9] Enrique Alba, Gabriel Luque, and Sergio Nesmachnow. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[10] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, Debra Hensgen, and Sahra Ali. Representing task and machine heterogeneities for heterogeneous computing systems. *Tamkang Journal of Science and Engineering*, 3(3):195–208, 2000.

[11] Enrico Angelelli, Renata Mansini, and M Grazia Speranza. Kernel search: A general heuristic for the multi-dimensional knapsack problem. *Computers & Operations Research*, 37(11):2017–2026, 2010.

[12] John E Beasley. OR-Library: Distributing test problems by electronic mail. *The Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[13] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, September 2003.

[14] Ilhem Boussaid, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82 – 117, 2013.

[15] Vincent Boyer, Moussa Elkihel, and Didier El Baz. Heuristics for the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, 199(3):658–664, 2009.

[16] Mathieu Brévilliers, Omar Abdelkafi, Julien Lepagnot, and Lhassane Idoumghar. Fast hybrid BSA-DE-SA algorithm on GPU. In *Swarm Intelligence Based Optimization: Second International Conference, ICSIBO 2016, Mulhouse, France, June 13-14, 2016, Revised Selected Papers*, volume 10103, page 75. Springer, 2017.

[17] André R Brodtkorb, Trond R Hagen, and Martin L Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

[18] Libor Bukata, Přemysl Scha, and Zdeněk Hanzálek. Solving the resource constrained project scheduling problem using the parallel Tabu Search designed for the CUDA platform. *Journal of Parallel and Distributed Computing*, 77:58–68, 2015.

[19] Rainer E Burkard and Franz Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17(2):169–174, 1984.

[20] Ann M. Campbell and Barrett W. Thomas. Probabilistic traveling salesman problem with deadlines. *Transportation Science*, 42(1):1–21, 2008.

[21] José M Cecilia, José M García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[22] Ugur Cekmez, Mustafa Ozsiginan, and Ozgur Koray Sahingoz. A UAV path planning with parallel ACO algorithm on CUDA platform. In *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, pages 347–354. IEEE, 2014.

[23] Paul C Chu and John E Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics*, 4(1):63–86, 1998.

[24] Pinar Civicioglu. Backtracking search optimization algorithm for numerical optimization problems. *Applied Mathematics and Computation*, 219(15):8121 – 8144, 2013.

[25] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[26] Daniele Costa and Alain Hertz. Ants can colour graphs. *Journal of the operational research society*, 48(3):295–305, 1997.

[27] Michał Czapiński. An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform. *Journal of Parallel and Distributed Computing*, 73(11):1461–1468, 2013.

[28] Narjess Dali and Sadok Bouamama. GPU-PSO: Parallel particle swarm optimization approaches on graphical processing unit for constraint reasoning: Case of MAX-CSPs. *Procedia Computer Science*, 60:1070–1080, 2015.

[29] Laurence Dawson and Iain A Stewart. Accelerating ant colony optimization-based edge detection on the GPU using CUDA. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 1736–1743. IEEE, 2014.

[30] Bianca de Almeida Dantas and Edson Norberto Cáceres. Sequential and parallel implementation of GRASP for the 0-1 Multidimensional Knapsack Problem. *Procedia Computer Science*, 51:2739 – 2743, 2015.

[31] Zvi Drezner, Peter M. Hahn, and Éeric D. Taillard. Recent advances for the quadratic assignment problem with special emphasis on instances that are difficult for meta-heuristic methods. *Annals of Operations Research*, 139(1):65–94, 2005.

[32] Sankha Baran Dutta, Robert McLeod, and Marcia Friesen. Gpu accelerated nature inspired methods for modelling large scale bi-directional pedestrian movement. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 448–456. IEEE, 2014.

[33] Wenbin Fang, Ka Keung Lau, Mian Lu, Xiangye Xiao, Chi Kit Lam, Philip Yang Yang, Bingsheng He, Qiong Luo, Pedro V Sander, and Ke Yang. Parallel data mining on graphics processors. *Hong Kong Univ. Sci. and Technology, Hong Kong, China, Tech. Rep. HKUST-CS08-07*, 2008.

[34] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing*, 71(10):3934–3957, 2015.

[35] Henrique Fingler, Edson N Cáceres, Henrique Mongelli, and Siang W Song. A CUDA based solution to the multidimensional knapsack problem using the ant colony optimization. *Procedia Computer Science*, 29:84–94, 2014.

[36] Wayne Franz and Parimala Thulasiraman. A dynamic cooperative hybrid MPSO + GA on hybrid CPU+ GPU fused multicore. In *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*, pages 1–8. IEEE, 2016.

[37] Fred Glover. Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[38] Fred Glover. A template for scatter search and path relinking. *Lecture notes in computer science*, 1363:13–54, 1998.

[39] Fred Glover and Manuel Laguna. Tabu search. In *Handbook of Combinatorial Optimization*, pages 3261–3362. Springer New York, 2013.

[40] F Herrera, M Lozano, and D Molina. Test suite for the special issue of soft computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems. 2010.

[41] Yue Hu, David M Koppelman, and Steven R Brandt. Thoroughly exploring gpu buffering options for stencil code by using an efficiency measure and a performance model. *IEEE Transactions on Multi-Scale Computing Systems*, 2017.

[42] Md Maruf Hussain, Hiroshi Hattori, and Noriyuki Fujimoto. A CUDA implementation of the standard particle swarm optimization. In *Symbolic and*

Numeric Algorithms for Scientific Computing (SYNASC), 2016 18th International Symposium on, pages 219–226. IEEE, 2016.

[43] Momin Jamil and Xin-She Yang. A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2):150–194, 2013.

[44] Min Jin and Huaxiang Lu. Parallel particle swarm optimization with genetic communication strategy and its implementation on GPU. In *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*, volume 1, pages 99–104. IEEE, 2012.

[45] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. An evolutionary approach to combinatorial optimization problems. In *ACM Conference on Computer Science*, pages 66–73. Citeseer, 1994.

[46] David Kirk et al. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[47] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[48] Pavel Krömer, Jan Platoš, and Václav Snášel. Nature-inspired meta-heuristics on modern GPUs: State of the art and brief survey of selected algorithms. *International Journal of Parallel Programming*, 42(5):681–709, 2014.

[49] Jitendra Kumar, Lotika Singh, and Sandeep Paul. GPU based parallel cooperative particle swarm optimization using C-CUDA: a case study. In *Fuzzy Systems (FUZZ), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[50] T. Lan, M. Guo, J. Qu, Y. Chai, Z. Liu, and X. Zhang. CUDA-based hierarchical multi-block particle swarm optimization algorithm. In *The 27th Chinese Control and Decision Conference (2015 CCDC)*, pages 4419–4423, May 2015.

[51] Miguel Lastra, Daniel Molina, and José M Benítez. A high performance memetic algorithm for extremely high-dimensional problems. *Information Sciences*, 293:35–58, 2015.

[52] Jianming Li, Wei Wang, and Xiangpei Hu. Parallel particle swarm optimization algorithm based on CUDA in the AWS cloud. In *Frontier of Computer Science and Technology (FCST), 2015 Ninth International Conference on*, pages 8–12. IEEE, 2015.

[53] Guo-Heng Luo, Sheng-Kai Huang, Yue-Shan Chang, and Shyan-Ming Yuan. A parallel bees algorithm implementation on GPU. *Journal of Systems Architecture*, 60(3):271–279, 2014.

[54] Sayyed Ali Mirsoleimani, Ali Karami, and Farshad Khunjush. A parallel memetic algorithm on GPU to solve the task scheduling problem in heterogeneous environments. pages 1181–1188, 2013.

[55] Javad Mohammadi, Kamal Mirzaie, and Vali Derhami. Parallel genetic algorithm based on GPU for solving quadratic assignment problem. In *2015 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, pages 569–572, Nov 2015.

[56] Ryouhei Murooka, Yasuaki Ito, and Koji Nakano. Accelerating ant colony optimization for the vertex coloring problem on the GPU. In *Computing and Networking (CANDAR), 2016 Fourth International Symposium on*, pages 469–475. IEEE, 2016.

[57] Hossein Nezamabadi-pour, Saeid Saryazdi, and Esmat Rashedi. Edge detection using ant algorithms. *Soft Computing*, 10(7):623–628, 2006.

[58] John Nickolls. Scalable parallel programming with CUDA introduction. In *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–9, Aug 2008.

[59] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens.

Parallel lossless data compression on the GPU. IEEE, 2012.

[60] Martín Pedemonte, Francisco Luna, and Enrique Alba. Systolic genetic search, a systolic computing-based metaheuristic. *Soft Computing*, 19(7):1779–1801, 2015.

[61] Duc Truong Pham and Michele Castellani. The bees algorithm: Modelling foraging behaviour to solve continuous optimization problems. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 223(12):2919–2938, 2009.

[62] Marcin Pietroń, Aleksander Byrski, and Marek Kisiel-Dorohinicki. GPGPU for difficult black-box problems. *Procedia Computer Science*, 51:1023–1032, 2015.

[63] Marcin Pietroń, Aleksander Byrski, and Marek Kisiel-Dorohinicki. Leveraging heterogeneous parallel platform in solving hard discrete optimization problems with metaheuristics. *Journal of Computational Science*, 18:59–68, 2017.

[64] Frédéric Pinel, Bernabé Dorronsoro, and Pascal Bouvry. Solving very large instances of the scheduling of independent tasks problem on the GPU. *Journal of Parallel and Distributed Computing*, 73(1):101–110, 2013.

[65] Yukihide Sasamura and Akihiro Fujiwara. A bee colony optimization algorithm for a connected sensor cover on GPGPU. In *2014 Second International Symposium on Computing and Networking*, pages 582–585, Dec 2014.

[66] Yuhui Shi and Russell Eberhart. A modified particle swarm optimizer. In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pages 69–73, May 1998.

[67] Rashmi Sharan Sinha, Satvir Singh, Sarabjeet Singh, and Vijay Kumar Banga. Speedup genetic algorithm using C-CUDA. In *2015 Fifth International Conference on Communication Systems and Network Technologies*, pages 1355–1359, April 2015.

[68] Rafał Skinderowicz. The GPU-based parallel ant colony system. *Journal of Parallel and Distributed Computing*, 98:48–60, 2016.

[69] Barbara M. Smith. The phase transition in constraint satisfaction problems: A closer look at the mushy region. In *Artificial Intelligence*, 1993.

[70] Francisco J. Solis and Roger J.-B. Wets. Minimization by random search techniques. *Mathematics of Operations Research*, 6(1):19–30, 1981.

[71] Jagiełło Szymon and Żelazny Dominik. Solving multi-criteria vehicle routing problem by parallel tabu search on GPU. *Procedia Computer Science*, 18:2529–2532, 2013.

[72] Éric Taillard. Paper: Robust taboo search for the quadratic assignment problem. *Parallel Comput.*, 17(4-5):443–455, July 1991.

[73] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009, 624 pages.

[74] Ke Tang, XD Li, PN Suganthan, ZY Yang, and Thomas Weise. Benchmark functions for the CEC 2010 special session and competition on large-scale global optimization (2010). *Nature Inspired Computation and Applications Laboratory, USTC, China*.

[75] Ke Tang, Xın Yáo, Ponnuthurai Nagaratnam Suganthan, Cara MacNish, Ying-Ping Chen, Chih-Ming Chen, and Zhenyu Yang. Benchmark functions for the CEC 2008 special session and competition on large scale global optimization. *Nature Inspired Computation and Applications Laboratory, USTC, China*, pages 153–177, 2007.

[76] Hamid R Tizhoosh. Opposition-based learning: A new scheme for machine intelligence. In *International Conference on Computational Intelligence for Mod-*

elling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06), volume 1, pages 695–701, Nov 2005.

[77] Christos Tsotskas, Timoleon Kipouros, and Anthony Mark Savill. The design and implementation of a GPU-enabled multi-objective tabu-search intended for real world and high-dimensional applications. *Procedia Computer Science*, 29:2152 – 2161, 2014.

[78] Roberto Ugolotti, Youssef SG Nashed, Pablo Mesejo, ŠPela Ivekovič, Luca Mussi, and Stefano Cagnoni. Particle swarm optimization and differential evolution for model-based object detection. *Applied Soft Computing*, 13(6):3092–3105, 2013.

[79] Pablo Vidal, Enrique Alba, and Francisco Luna. Solving optimization problems using a hybrid systolic search on GPU plus CPU. *Soft Computing*, pages 1–19, 2016.

[80] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 191–202. IEEE Press, 2014.

[81] Kai-Cheng Wei, Chao-chin Wu, and Chien-Ju Wu. Using CUDA GPU to accelerate the ant colony optimization algorithm. In *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 90–95, Dec 2013.

[82] Kai-Cheng Wei, Chao-Chin Wu, and Hui-Liang Yu. Mapping the simulated annealing algorithm onto CUDA GPUs. In *2015 10th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*, pages 358–365, Nov 2015.

[83] Hans Martin Weingartner and David N. Ness. Methods for the solution of the multidimensional 0/1 knapsack problem. *Oper. Res.*, 15(1):83–103, February 1967.

[84] Yun Wen, Hua Xu, and Jiadong Yang. A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system. *Information Sciences*, 181(3):567 – 581, 2011.

[85] Dennis Weyland, Roberto Montemanni, and Luca Maria Gambardella. A metaheuristic framework for stochastic combinatorial optimization problems based on GPGPU with a case study on the probabilistic traveling salesman problem with deadlines. *Journal of Parallel and Distributed Computing*, 73(1):74–85, 2013.

[86] Darrell Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[87] Zhiyi Yang, Yating Zhu, and Yong Pu. Parallel image processing based on CUDA. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 3, pages 198–201. IEEE, 2008.

[88] Hassan Youness, Aziza Ibraheim, Mohammed Moness, and Muhammad Osama. An efficient implementation of ant colony optimization on GPU for the satisfiability problem. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 230–235, March 2015.

[89] Drahoslav Zan and Jiri Jaros. Solving the multidimensional knapsack problem using a CUDA accelerated PSO. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2933–2939, July 2014.

[90] Xinyu Zhou, Zhijian Wu, and Hui Wang. Elite opposition-based differential evolution for solving large-scale optimization problems and its implementation on GPU. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 727–732, Dec 2012.